Python 正则表达式操作指南

wizardforcel

Published with GitBook

目錄

Python正则表达式操作指南	0
简介	1
简单模式	2
字符匹配	2.1
重复	2.2
使用正则表达式	3
编译正则表达式	3.1
反斜杠的麻烦	3.2
执行匹配	3.3
模块级函数	3.4
编译标志	3.5
更多模式功能	4
更多的元字符	4.1
分组	4.2
无捕获组和命名组	4.3
前向界定符	4.4
修改字符串	5
将字符串分片	5.1
搜索和替换	5.2
常见问题	6
使用字符串方式	6.1
match() vs search()	6.2
贪婪 vs 不贪婪	6.3
不用 re.VERBOSE	6.4
反馈	7

Python正则表达式操作指南

出自Ubuntu中文

原文出处：http://www.amk.ca/python/howto/regex/

原文作者：A.M. Kuchling （amk@amk.ca）

授权许可：创作共用协议

翻译人员：FireHare

校对人员：Leal

适用版本：Python 1.5 及后续版本

摘要

本文是通过Python的 re 模块来使用正则表达式的一个入门教程，和库参考手册的对应章节相比，更为浅显易懂、循序渐进。

本文可以从 http://www.amk.ca/python/howto 捕获

简介

Python 自1.5版本起增加了re 模块，它提供 Perl 风格的正则表达式模式。Python 1.5之前版本则是通过 regex 模块提供 Emacs 风格的模式。Emacs 风格模式可读性稍差些，而且功能也不强，因此编写新代码时尽量不要再使用 regex 模块，当然偶尔你还是可能在老代码里发现其踪影。

就其本质而言，正则表达式（或 RE）是一种小型的、高度专业化的编程语言，（在Python中）它内嵌在Python中，并通过 re 模块实现。使用这个小型语言，你可以为想要匹配的相应字符串集指定规则；该字符串集可能包含英文语句、e-mail地址、TeX命令或任何你想搞定的东西。然后你可以问诸如“这个字符串匹配该模式吗？”或“在这个字符串中是否有部分匹配该模式呢？”。你也可以使用 RE 以各种方式来修改或分割字符串。

正则表达式模式被编译成一系列的字节码，然后由用 C 编写的匹配引擎执行。在高级用法中，也许还要仔细留意引擎是如何执行给定 RE ，如何以特定方式编写 RE 以令生产的字节码运行速度更快。本文并不涉及优化，因为那要求你已充分掌握了匹配引擎的内部机制。

正则表达式语言相对小型和受限（功能有限），因此并非所有字符串处理都能用正则表达式完成。当然也有些任务可以用正则表达式完成，不过最终表达式会变得异常复杂。碰到这些情形时，编写 Python 代码进行处理可能反而更好；尽管 Python 代码比一个精巧的正则表达式要慢些，但它更易理解。

简单模式

我们将从最简单的正则表达式学习开始。由于正则表达式常用于字符串操作，那我们就从最常见的任务：字符匹配 下手。

有关正则表达式底层的计算机科学上的详细解释（确定性和非确定性有限自动机），你可以查阅编写编译器相关的任何教科书。

字符匹配

大多数字母和字符一般都会和自身匹配。例如，正则表达式 test 会和字符串“test”完全匹配。（你也可以使用大小写不敏感模式，它还能让这个 RE 匹配“Test”或“TEST”；稍后会有更多解释。）

这个规则当然会有例外；有些字符比较特殊，它们和自身并不匹配，而是会表明应和一些特殊的东西匹配，或者它们会影响到 RE 其它部分的重复次数。本文很大篇幅专门讨论了各种元字符及其作用。

这里有一个元字符的完整列表；其含义会在本指南余下部分进行讨论。

```
. ^ $ * + ? { [ ] \ | ( )
```

我们首先考察的元字符是"[" 和 "]"。它们常用来指定一个字符类别，所谓字符类别就是你想匹配的一个字符集。字符可以单个列出，也可以用“-”号分隔的两个给定字符来表示一个字符区间。例如，[abc] 将匹配"a", "b", 或 "c"中的任意一个字符；也可以用区间[a-c]来表示同一字符集，和前者效果一致。如果你只想匹配小写字母，那么 RE 应写成 [a-z].

元字符在类别里并不起作用。例如，[akm$]将匹配字符"a", "k", "m", 或 "$" 中的任意一个；"$"通常用作元字符，但在字符类别里，其特性被除去，恢复成普通字符。

你可以用补集来匹配不在区间范围内的字符。其做法是把"^"作为类别的首个字符；其它地方的"^"只会简单匹配 "^"字符本身。例如，5 将匹配除 "5" 之外的任意字符。

也许最重要的元字符是反斜杠"\"。 做为 Python 中的字符串字母，反斜杠后面可以加不同的字符以表示不同特殊意义。它也可以用于取消所有的元字符，这样你就可以在模式中匹配它们了。举个例子，如果你需要匹配字符 "[" 或 "\"，你可以在它们之前用反斜杠来取消它们的特殊意义：[或 \。

一些用 "\" 开始的特殊字符所表示的预定义字符集通常是很有用的，象数字集，字母集，或其它非空字符集。下列是可用的预设特殊字符：

```
\d  匹配任何十进制数；它相当于类 [0-9]。
\D  匹配任何非数字字符；它相当于类 [^0-9]。
\s  匹配任何空白字符；它相当于类  [ \t\n\r\f\v]。
\S  匹配任何非空白字符；它相当于类 [^ \t\n\r\f\v]。
\w  匹配任何字母数字字符；它相当于类 [a-zA-Z0-9_]。
\W  匹配任何非字母数字字符；它相当于类 [^a-zA-Z0-9_]。
```

这样特殊字符都可以包含在一个字符类中。如，[\s,.]字符类将匹配任何空白字符或","或"."。

本节最后一个元字符是 . 。它匹配除了换行字符外的任何字符，在 alternate 模式（re.DOTALL）下它甚至可以匹配换行。"." 通常被用于你想匹配“任何字符”的地方。

重复

正则表达式第一件能做的事是能够匹配不定长的字符集，而这是其它能作用在字符串上的方法所不能做到的。 不过，如果那是正则表达式唯一的附加功能的话，那么它们也就不那么优秀了。它们的另一个功能就是你可以指定正则表达式的一部分的重复次数。

我们讨论的第一个重复功能的元字符是 。 并不匹配字母字符 "*"；相反，它指定前一个字符可以被匹配零次或更多次，而不是只有一次。

举个例子，ca*t 将匹配 "ct" (0 个 "a" 字符), "cat" (1 个 "a"), "caaat" (3 个 "a" 字符)等等。RE 引擎有各种来自 C 的整数类型大小的内部限制，以防止它匹配超过20亿个 "a" 字符；你也许没有足够的内存去建造那么大的字符串，所以将不会累计到那个限制。

象 * 这样地重复是“贪婪的”；当重复一个 RE 时，匹配引擎会试着重复尽可能多的次数。如果模式的后面部分没有被匹配，匹配引擎将退回并再次尝试更小的重复。

一步步的示例可以使它更加清晰。让我们考虑表达式 a[bcd]*b。它匹配字母 "a"，零个或更多个来自类 [bcd]中的字母，最后以 "b" 结尾。现在想一想该 RE 对字符串 "abcbd" 的匹配。

Step	Matched	Explanation
1	a	a 匹配模式
2	abcbd	引擎匹配 [bcd]*，并尽其所能匹配到字符串的结尾
3	Failure	引擎尝试匹配 b，但当前位置已经是字符的最后了，所以失败
4	abcb	退回，[bcd]*尝试少匹配一个字符。
5	Failure	再次尝次b，但在当前最后一位字符是"d"。
6	abc	再次退回，[bcd]*只匹配 "bc"。
7	abcb	再次尝试 b ，这次当前位上的字符正好是 "b"

RE 的结尾部分现在可以到达了，它匹配 "abcb"。这证明了匹配引擎一开始会尽其所能进行匹配，如果没有匹配然后就逐步退回并反复尝试 RE 剩下来的部分。直到它退回尝试匹配 [bcd] 到零次为止，如果随后还是失败，那么引擎就会认为该字符串根本无法匹配 RE 。

另一个重复元字符是 +，表示匹配一或更多次。请注意 和 + 之间的不同； 匹配零或更多次，所以可以根本就不出现，而 + 则要求至少出现一次。用同一个例子，ca+t 就可以匹配 "cat" (1 个 "a")， "caaat" (3 个 "a")， 但不能匹配 "ct"。

还有更多的限定符。问号 ? 匹配一次或零次；你可以认为它用于标识某事物是可选的。例如：home-?brew 匹配 "homebrew" 或 "home-brew"。

最复杂的重复限定符是 {m,n}(注意m,n之间不能有空格)，其中 m 和 n 是十进制整数。该限定符的意思是至少有 m 个重复，至多到 n 个重复。举个例子，a/{1,3}b 将匹配 "a/b"，"a//b" 和 "a///b"。它不能匹配 "ab" 因为没有斜杠，也不能匹配 "a////b" ，因为有四个。

你可以忽略 m 或 n；因为会为缺失的值假设一个合理的值。忽略 m 会认为下边界是 0，而忽略 n 的结果将是上边界为无穷大 -- 实际上是先前我们提到的20亿，但这也许同无穷大一样。

细心的读者也许注意到其他三个限定符都可以用这样方式来表示。 {0,} 等同于 ，*{1,}* 等同于 +，而*{0,1}*则与 *?* 相同。如果可以的话，最好使用 ，+，或?。很简单因为它们更短也更容易懂。

使用正则表达式

现在我们已经看了一些简单的正则表达式，那么我们实际在 Python 中是如何使用它们的呢？re 模块提供了一个正则表达式引擎的接口，可以让你将 REs 编译成对象并用它们来进行匹配。

编译正则表达式

正则表达式被编译成 `RegexObject` 实例，可以为不同的操作提供方法，如模式匹配搜索或字符串替换。

```
#python
>>> import re
>>> p = re.compile('ab*')
>>> print p
<_sre.SRE_Pattern object at 0xb76e1a70>
```

re.compile() 也接受可选的标志参数，常用来实现不同的特殊功能和语法变更。我们稍后将查看所有可用的设置，但现在只举一个例子：

```
#!python
>>> p = re.compile('ab*', re.IGNORECASE)
```

RE 被做为一个字符串发送给 re.compile()。REs 被处理成字符串是因为正则表达式不是 Python 语言的核心部分，也没有为它创建特定的语法。（应用程序根本就不需要 REs，因此没必要包含它们去使语言说明变得臃肿不堪。）而 re 模块则只是以一个 C 扩展模块的形式来被 Python 包含，就象 socket 或 zlib 模块一样

将 REs 作为字符串以保证 Python 语言的简洁，但这样带来的一个麻烦就是象下节标题所讲的。

反斜杠的麻烦

在早期规定中，正则表达式用反斜杠字符 ("\") 来表示特殊格式或允许使用特殊字符而不调用它的特殊用法。这就与 Python 在字符串中的那些起相同作用的相同字符产生了冲突。

让我们举例说明，你想写一个 RE 以匹配字符串 "\section"，可能是在一个 LATEX 文件查找。为了要在程序代码中判断，首先要写出想要匹配的字符串。接下来你需要在所有反斜杠和其它元字符前加反斜杠来取消其特殊意义，结果要匹配的字符串就成了"\section"。 当把这个字符串传递给re.compile()时必须还是"\section"。然而，作为Python的字符串实值(string literals)来表示的话，"\section"中两个反斜杠还要再次取消特殊意义，最后结果就变成了"\\section"。

字符	阶段
\section	要匹配的字符串
\section	为 re.compile 取消反斜杠的特殊意义
"\\section"	为"\section"的字符串实值(string literals)取消反斜杠的特殊意义

简单地说，为了匹配一个反斜杠，不得不在 RE 字符串中写 '\\'，因为正则表达式中必须是 "\"，而每个反斜杠在常规的 Python 字符串实值中必须表示成 "\"。在 REs 中反斜杠的这个重复特性会导致大量重复的反斜杠，而且所生成的字符串也很难懂。

解决的办法就是为正则表达式使用 Python 的 raw 字符串表示；在字符串前加个 "r" 反斜杠就不会被任何特殊方式处理，所以 r"\n" 就是包含"\" 和 "n" 的两个字符，而 "\n" 则是一个字符，表示一个换行。正则表达式通常在 Python 代码中都是用这种 raw 字符串表示。

常规字符串	**Raw** 字符串
"ab*"	r"ab*"
"\\section"	r"\section"
"\w+\s+\1"	r"\w+\s+\1"

执行匹配

一旦你有了已经编译了的正则表达式的对象，你要用它做什么呢？`RegexObject` 实例有一些方法和属性。这里只显示了最重要的几个，如果要看完整的列表请查阅 Python Library Reference

方法/属性	作用
match()	决定 RE 是否在字符串刚开始的位置匹配
search()	扫描字符串，找到这个 RE 匹配的位置
findall()	找到 RE 匹配的所有子串，并把它们作为一个列表返回
finditer()	找到 RE 匹配的所有子串，并把它们作为一个迭代器返回

如果没有匹配到的话，match() 和 search() 将返回 None。如果成功的话，就会返回一个 `MatchObject` 实例，其中有这次匹配的信息：它是从哪里开始和结束，它所匹配的子串等等。

你可以用采用人机对话并用 re 模块实验的方式来学习它。如果你有 Tkinter 的话，你也许可以考虑参考一下 Tools/scripts/redemo.py，一个包含在 Python 发行版里的示范程序。

首先，运行 Python 解释器，导入 re 模块并编译一个 RE：

```
#!python
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 80c3c28>
```

现在，你可以试着用 RE 的 [a-z]+ 去匹配不同的字符串。一个空字符串将根本不能匹配，因为 + 的意思是“一个或更多的重复次数”。 在这种情况下 match() 将返回 None，因为它使解释器没有输出。你可以明确地打印出 match() 的结果来弄清这一点。

```
#!python
>>> p.match("")
>>> print p.match("")
None
```

现在，让我们试着用它来匹配一个字符串，如 "tempo"。这时，match() 将返回一个 MatchObject。因此你可以将结果保存在变量里以便後面使用。

```
#!python
>>> m = p.match( 'tempo')
>>> print m
<_sre.SRE_Match object at 80c4f68>
```

现在你可以查询 `MatchObject` 关于匹配字符串的相关信息了。MatchObject 实例也有几个方法和属性；最重要的那些如下所示：

方法/属性	作用
group()	返回被 RE 匹配的字符串
start()	返回匹配开始的位置
end()	返回匹配结束的位置
span()	返回一个元组包含匹配 (开始,结束) 的位置

试试这些方法不久就会清楚它们的作用了：

```
#!python
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

group() 返回 RE 匹配的子串。start() 和 end() 返回匹配开始和结束时的索引。span() 则用单个元组把开始和结束时的索引一起返回。因为匹配方法检查到如果 RE 在字符串开始处开始匹配，那么 start() 将总是为零。然而， `RegexObject` 实例的 search 方法扫描下面的字符串的话，在这种情况下，匹配开始的位置就也许不是零了。

```
#!python
>>> print p.match('::: message')
None
>>> m = p.search('::: message') ; print m
<re.MatchObject instance at 80c9650>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

在实际程序中，最常见的作法是将 `MatchObject` 保存在一个变量里，然後检查它是否为 None，通常如下所示：

```
#!python
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
print 'Match found: ', m.group()
else:
print 'No match'
```

两个 `RegexObject` 方法返回所有匹配模式的子串。findall()返回一个匹配字符串行表：

```
#!python
>>> p = re.compile('\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

findall() 在它返回结果时不得不创建一个列表。在 Python 2.2中，也可以用 finditer() 方法。

```
#!python
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x401833ac>
>>> for match in iterator:
...     print match.span()
...
(0, 2)
(22, 24)
(29, 31)
```

模块级函数

你不一定要产生一个 `RegexObject` 对象然后再调用它的方法；re 模块也提供了顶级函数调用如 match()、search()、sub() 等等。这些函数使用 RE 字符串作为第一个参数，而后面的参数则与相应 `RegexObject` 的方法参数相同，返回则要么是 None 要么就是一个 `MatchObject` 的实例。

```
#!python
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.MatchObject instance at 80c5978>
```

Under the hood, 这些函数简单地产生一个 RegexOject 并在其上调用相应的方法。它们也在缓存里保存编译后的对象，因此在将来调用用到相同 RE 时就会更快。

你将使用这些模块级函数，还是先得到一个 `RegexObject` 再调用它的方法呢？如何选择依赖于怎样用 RE 更有效率以及你个人编码风格。如果一个 RE 在代码中只做用一次的话，那么模块级函数也许更方便。如果程序包含很多的正则表达式，或在多处复用同一个的话，那么将全部定义放在一起，在一段代码中提前编译所有的 REs 更有用。从标准库中看一个例子，这是从 xmllib.py 文件中提取出来的：

```
#!python
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

我通常更喜欢使用编译对象，甚至它只用一次，但很少人会像我这样做(如同一个纯粹主义者)。

编译标志

编译标志让你可以修改正则表达式的一些运行方式。在 re 模块中标志可以使用两个名字，一个是全名如 IGNORECASE，一个是缩写，一字母形式如 I。（如果你熟悉 Perl 的模式修改，一字母形式使用同样的字母；例如 re.VERBOSE的缩写形式是 re.X。）多个标志可以通过按位 OR-ing 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：

这有个可用标志表，对每个标志后面都有详细的说明。

标志	含义
DOTALL, S	使 . 匹配包括换行在内的所有字符
IGNORECASE, I	使匹配对大小写不敏感
LOCALE, L	做本地化识别（locale-aware）匹配
MULTILINE, M	多行匹配，影响 ^ 和 $
VERBOSE, X	能够使用 REs 的 verbose 状态，使之被组织得更清晰易懂

I IGNORECASE

使匹配对大小写不敏感；字符类和字符串匹配字母时忽略大小写。举个例子，[A-Z]也可以匹配小写字母，Spam 可以匹配 "Spam", "spam", 或 "spAM"。这个小写字母并不考虑当前位置。

L LOCALE

影响 \w, \W, \b, 和 \B，这取决于当前的本地化设置。

locales 是 C 语言库中的一项功能，是用来为需要考虑不同语言的编程提供帮助的。举个例子，如果你正在处理法文文本，你想用 \w+ 来匹配文字，但 \w 只匹配字符类 [A-Za-z]；它并不能匹配 "é" 或 "ç"。如果你的系统配置适当且本地化设置为法语，那么内部的 C 函数将告诉程序 "é" 也应该被认为是一个字母。当在编译正则表达式时使用 LOCALE 标志会得到用这些 C 函数来处理 \w 后的编译对象；这会更慢，但也会象你希望的那样可以用 \w+ 来匹配法文文本。

M MULTILINE

(此时 ^ 和 $ 不会被解释; 它们将在 4.1 节被介绍.)

使用 "^" 只匹配字符串的开始，而 $ 则只匹配字符串的结尾和直接在换行前（如果有的话）的字符串结尾。当本标志指定后， "^" 匹配字符串的开始和字符串中每行的开始。同样的， $ 元字符匹配字符串结尾和字符串中每行的结尾（直接在每个换行之前）。

S DOTALL

使 "." 特殊字符完全匹配任何字符，包括换行；没有这个标志， "." 匹配除了换行外的任何字符。

X VERBOSE

该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。当该标志被指定时，在 RE 字符串中的空白符被忽略，除非该空白符在字符类中或在反斜杠之后；这可以让你更清晰地组织和缩进 RE。它也可以允许你将注释写入 RE，这些注释会被引擎忽略；注释用 "#"号 来标识，不过该符号不能在字符串或反斜杠之后。

举个例子，这里有一个使用 re.VERBOSE 的 RE；看看读它轻松了多少？

```
#!python
charref = re.compile(r"""&[[]]           # Start of a numeric entity reference|||here has
(
[0-9]+[^0-9]      # Decimal form
| 0[0-7]+[^0-7]   # Octal form
| x[0-9a-fA-F]+[^0-9a-fA-F] # Hexadecimal form
)
""", re.VERBOSE)
```

没有 verbose 设置， RE 会看起来象这样：

```
#!python
charref = re.compile("&#([0-9]+[^0-9]"
"|0[0-7]+[^0-7]"
"|x[0-9a-fA-F]+[^0-9a-fA-F])")
```

在上面的例子里，Python 的字符串自动连接可以用来将 RE 分成更小的部分，但它比用 re.VERBOSE 标志时更难懂

更多模式功能

到目前为止，我们只展示了正则表达式的一部分功能。在本节，我们将展示一些新的元字符和如何使用组来检索被匹配的文本部分。

== ==

更多的元字符

还有一些我们还没展示的元字符，其中的大部分将在本节展示。

剩下来要讨论的一部分元字符是零宽界定符（zero-width assertions）。它们并不会使引擎在处理字符串时更快;相反，它们根本就没有对应任何字符，只是简单的成功或失败。举个例子， \b 是一个在单词边界定位当前位置的界定符（assertions），这个位置根本就不会被 \b 改变。这意味着零宽界定符（zero-width assertions）将永远不会被重复，因为如果它们在给定位置匹配一次，那么它们很明显可以被匹配无数次。

|

可选项，或者 "or" 操作符。如果 A 和 B 是正则表达式，A|B 将匹配任何匹配了 "A" 或 "B" 的字符串。| 的优先级非常低，是为了当你有多字符串要选择时能适当地运行。Crow|Servo 将匹配"Crow" 或 "Servo", 而不是 "Cro", 一个 "w" 或 一个 "S", 和 "ervo"。

为了匹配字母 "|"，可以用 |，或将其包含在字符类中，如[|]。

^

匹配行首。除非设置 MULTILINE 标志，它只是匹配字符串的开始。在 MULTILINE 模式里，它也可以直接匹配字符串中的每个换行。

例如，如果你只希望匹配在行首单词 "From"，那么 RE 将用 ^From。

```
#!python
>>> print re.search('^From', 'From Here to Eternity')
<re.MatchObject instance at 80c1520>
>>> print re.search('^From', 'Reciting From Memory')
None
```

$

匹配行尾，行尾被定义为要么是字符串尾，要么是一个换行字符后面的任何位置。

```
#!python
>>> print re.search('}$', '{block}')
<re.MatchObject instance at 80adfa8>
>>> print re.search('}$', '{block} ')
None
>>> print re.search('}$', '{block}\n')
<re.MatchObject instance at 80adfa8>
```

匹配一个 "$"，使用 $ 或将其包含在字符类中，如[$]。

\A

只匹配字符串首。当不在 MULTILINE 模式，\A 和 ^ 实际上是一样的。然而，在 MULTILINE 模式里它们是不同的；\A 只是匹配字符串首，而 ^ 还可以匹配在换行符之后字符串的任何位置。

\Z

Matches only at the end of the string. 只匹配字符串尾。

\b

单词边界。这是个零宽界定符（zero-width assertions）只用以匹配单词的词首和词尾。单词被定义为一个字母数字序列，因此词尾就是用空白符或非字母数字符来标示的。

下面的例子只匹配 "class" 整个单词；而当它被包含在其他单词中时不匹配。

```
#!python
>>> p = re.compile(r'\bclass\b')
>>> print p.search('no class at all')
<re.MatchObject instance at 80c8f28>
>>> print p.search('the declassified algorithm')
None
>>> print p.search('one subclass is')
None
```

当用这个特殊序列时你应该记住这里有两个微妙之处。第一个是 Python 字符串和正则表达式之间最糟的冲突。在 Python 字符串里，"\b" 是反斜杠字符，ASCII值是8。如果你没有使用 raw 字符串时，那么 Python 将会把 "\b" 转换成一个回退符，你的 RE 将无法象你希望的那样匹配它了。下面的例子看起来和我们前面的 RE 一样，但在 RE 字符串前少了一个 "r" 。

```
#!python
>>> p = re.compile('\bclass\b')
>>> print p.search('no class at all')
None
>>> print p.search('\b' + 'class' + '\b')
<re.MatchObject instance at 80c3ee0>
```

第二个在字符类中，这个限定符（assertion）不起作用，\b 表示回退符，以便与 Python 字符串兼容。

\B

另一个零宽界定符（zero-width assertions），它正好同 \b 相反，只在当前位置不在单词边界时匹配。

分组

你经常需要得到比 RE 是否匹配还要多的信息。正则表达式常常用来分析字符串，编写一个 RE 匹配感兴趣的部分并将其分成几个小组。举个例子，一个 RFC-822 的头部用 ":" 隔成一个头部名和一个值，这就可以通过编写一个正则表达式匹配整个头部，用一组匹配头部名，另一组匹配头部值的方式来处理。

组是通过 "(" 和 ")" 元字符来标识的。 "(" 和 ")" 有很多在数学表达式中相同的意思；它们一起把在它们里面的表达式组成一组。举个例子，你可以用重复限制符，象 *, +, ?, 和 {m,n}，来重复组里的内容，比如说(ab)* 将匹配零或更多个重复的 "ab"。

```
#!python
>>> p = re.compile('(ab)*')
>>> print p.match('ababababab').span()
(0, 10)
```

组用 "(" 和 ")" 来指定，并且得到它们匹配文本的开始和结尾索引；这就可以通过一个参数用 group()、start()、end() 和 span() 来进行检索。组是从 0 开始计数的。组 0 总是存在；它就是整个 RE，所以 `MatchObject` 的方法都把组 0 作为它们缺省的参数。稍后我们将看到怎样表达不能得到它们所匹配文本的 span。

```
#!python
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

小组是从左向右计数的，从1开始。组可以被嵌套。计数的数值可以通过从左到右计算打开的括号数来确定。

```
#!python
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。

```
#!python
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

The groups() 方法返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。

```
#!python
>>> m.groups()
('abc', 'b')
```

模式中的逆向引用允许你指定先前捕获组的内容，该组也必须在字符串当前位置被找到。举个例子，如果组 1 的内容能够在当前位置找到的话，\1 就成功否则失败。记住 Python 字符串也是用反斜杠加数据来允许字符串中包含任意字符的，所以当在 RE 中使用逆向引用时确保使用 raw 字符串。

例如，下面的 RE 在一个字符串中找到成双的词。

```
#!python
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

象这样只是搜索一个字符串的逆向引用并不常见 -- 用这种方式重复数据的文本格式并不多见 -- 但你不久就可以发现它们用在字符串替换上非常有用。

无捕获组和命名组

精心设计的 REs 也许会用很多组，既可以捕获感兴趣的子串，又可以分组和结构化 RE 本身。在复杂的 REs 里，追踪组号变得困难。有两个功能可以对这个问题有所帮助。它们也都使用正则表达式扩展的通用语法，因此我们来看看第一个。

Perl 5 对标准正则表达式增加了几个附加功能，Python 的 re 模块也支持其中的大部分。选择一个新的单按键元字符或一个以 "\" 开始的特殊序列来表示新的功能，而又不会使 Perl 正则表达式与标准正则表达式产生混乱是有难度的。如果你选择 "&" 做为新的元字符，举个例子，老的表达式认为 "&" 是一个正常的字符，而不会在使用 \& 或 [&] 时也不会转义。

Perl 开发人员的解决方法是使用 (?...) 来做为扩展语法。"?" 在括号后面会直接导致一个语法错误，因为 "?" 没有任何字符可以重复，因此它不会产生任何兼容问题。紧随 "?" 之后的字符指出扩展的用途，因此 (?=foo)

Python 新增了一个扩展语法到 Perl 扩展语法中。如果在问号后的第一个字符是 "P"，你就可以知道它是针对 Python 的扩展。目前有两个这样的扩展: (?P<name>...) 定义一个命名组，(?P=name) 则是对命名组的逆向引用。如果 Perl 5 的未来版本使用不同的语法增加了相同的功能，那么 re 模块也将改变以支持新的语法，与此同时为了兼容性的目的而继续保持的 Python 专用语法。

现在我们看一下普通的扩展语法，我们回过头来简化在复杂 REs 中使用组运行的特性。因为组是从左到右编号的，而且一个复杂的表达式也许会使用许多组，它可以使跟踪当前组号变得困难，而修改如此复杂的 RE 是十分麻烦的。在开始时插入一个新组，你可以改变它之后的每个组号。

首先，有时你想用一个组去收集正则表达式的一部分，但又对组的内容不感兴趣。你可以用一个无捕获组: (?:...) 来实现这项功能，这样你可以在括号中发送任何其他正则表达式。

```
#!python
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

除了捕获匹配组的内容之外，无捕获组与捕获组表现完全一样；你可以在其中放置任何字符，可以用重复元字符如 "*" 来重复它，可以在其他组（无捕获组与捕获组）中嵌套它。(?:...) 对于修改已有组尤其有用，因为你可以不用改变所有其他组号的情况下添加一个新组。捕获组和无捕获组在搜索效率方面也没什么不同，没有哪一个比另一个更快。

其次，更重要和强大的是命名组；与用数字指定组不同的是，它可以用名字来指定。

命令组的语法是 Python 专用扩展之一： (?P<name>...)。名字很明显是组的名字。除了该组有个名字之外，命名组也同捕获组是相同的。 `MatchObject` 的方法处理捕获组时接受的要么是表示组号的整数，要么是包含组名的字符串。命名组也可以是数字，所以你可以通过两种方式来得到一个组的信息：

```
#!python
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

命名组是便于使用的，因为它可以让你使用容易记住的名字来代替不得不记住的数字。这里有一个来自 imaplib 模块的 RE 示例：

```
#!python
InternalDate = re.compile(r'INTERNALDATE "'
r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
    r'(?P<year>[0-9][0-9][0-9][0-9])'
r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
r'"')
```

很明显，得到 m.group('zonem') 要比记住得到组 9 要容易得多。

因为逆向引用的语法，象 (...)\1 这样的表达式所表示的是组号，这时用组名代替组号自然会有差别。还有一个 Python 扩展：(?P=name) ，它可以使叫 name 的组内容再次在当前位置发现。正则表达式为了找到重复的单词，(\b\w+)\s+\1 也可以被写成 (?P<word>\b\w+)\s+(?P=word)：

```
#!python
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

前向界定符

另一个零宽界定符（zero-width assertion）是前向界定符。前向界定符包括前向肯定界定符和前项否定界定符，如下所示：

(?=...)

前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。

(?!...)

前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功

通过示范在哪前向可以成功有助于具体实现。考虑一个简单的模式用于匹配一个文件名，并将其通过 "." 分成基本名和扩展名两部分。如在 "news.rc" 中，"news" 是基本名，"rc" 是文件的扩展名。

匹配模式非常简单：

```
.*[.].*$
```

注意 "." 需要特殊对待，因为它是一个元字符；我把它放在一个字符类中。另外注意后面的 $; 添加这个是为了确保字符串所有的剩余部分必须被包含在扩展名中。这个正则表达式匹配 "foo.bar"、"autoexec.bat"、 "sendmail.cf" 和 "printers.conf"。

现在，考虑把问题变得复杂点；如果你想匹配的扩展名不是 "bat" 的文件名？一些不正确的尝试：

```
.*[.][^b].*$
```

上面的第一次去除 "bat" 的尝试是要求扩展名的第一个字符不是 "b"。这是错误的，因为该模式也不能匹配 "foo.bar"。

```
.*[.]([^b]..|.[^a].|..[^t])$
```

当你试着修补第一个解决方法而要求匹配下列情况之一时表达式更乱了：扩展名的第一个字符不是 "b"; 第二个字符不是 "a"；或第三个字符不是 "t"。这样可以接受 "foo.bar" 而拒绝 "autoexec.bat"，但这要求只能是三个字符的扩展名而不接受两个字符的扩展名如 "sendmail.cf"。我们将在努力修补它时再次把该模式变得复杂。

```
.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$
```

在第三次尝试中，第二和第三个字母都变成可选，为的是允许匹配比三个字符更短的扩展名，如 "sendmail.cf"。

该模式现在变得非常复杂，这使它很难读懂。更糟的是，如果问题变化了，你想扩展名不是 "bat" 和 "exe"，该模式甚至会变得更复杂和混乱。

前向否定把所有这些裁剪成：

```
.*[.](?!bat$).*$
```

前向的意思：如果表达式 bat 在这里没有匹配，尝试模式的其余部分；如果 bat$ 匹配，整个模式将失败。后面的 $ 被要求是为了确保象 "sample.batch" 这样扩展名以 "bat" 开头的会被允许。

将另一个文件扩展名排除在外现在也容易；简单地将其做为可选项放在界定符中。下面的这个模式将以 "bat" 或 "exe" 结尾的文件名排除在外。

```
.*[.](?!bat$|exe$).*$
```

修改字符串

到目前为止，我们简单地搜索了一个静态字符串。正则表达式通常也用不同的方式，通过下面的 `RegexObject` 方法，来修改字符串。

方法/属性	作用
split()	将字符串在 RE 匹配的地方分片并生成一个列表，
sub()	找到 RE 匹配的所有子串，并将其用一个不同的字符串替换
subn()	与 sub() 相同，但返回新的字符串和替换次数

将字符串分片

`RegexObject` 的 split() 方法在 RE 匹配的地方将字符串分片，将返回列表。它同字符串的 split() 方法相似但提供更多的定界符；split()只支持空白符和固定字符串。就象你预料的那样，也有一个模块级的 re.split() 函数。

```
split(string [, maxsplit = 0])
```

通过正则表达式将字符串分片。如果捕获括号在 RE 中使用，那么它们的内容也会作为结果列表的一部分返回。如果 maxsplit 非零，那么最多只能分出 maxsplit 个分片。

你可以通过设置 maxsplit 值来限制分片数。当 maxsplit 非零时，最多只能有 maxsplit 个分片，字符串的其余部分被做为列表的最后部分返回。在下面的例子中，定界符可以是非数字字母字符的任意序列。

```
#!python
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

有时，你不仅对定界符之间的文本感兴趣，也需要知道定界符是什么。如果捕获括号在 RE 中使用，那么它们的值也会当作列表的一部分返回。比较下面的调用：

```
#!python
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

模块级函数 re.split() 将 RE 作为第一个参数，其他一样。

```
#!python
>>> re.split('[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

搜索和替换

其他常见的用途就是找到所有模式匹配的字符串并用不同的字符串来替换它们。sub() 方法提供一个替换值，可以是字符串或一个函数，和一个要被处理的字符串。

```
sub(replacement, string[, count = 0])
```

返回的字符串是在字符串中用 RE 最左边不重复的匹配来替换。如果模式没有发现，字符将被没有改变地返回。

可选参数 count 是模式匹配后替换的最大次数；count 必须是非负整数。缺省值是 0 表示替换所有的匹配。

这里有个使用 sub() 方法的简单例子。它用单词 "colour" 替换颜色名。

```
#!python
>>> p = re.compile( '(blue|white|red)')
>>> p.sub( 'colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub( 'colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

subn() 方法作用一样，但返回的是包含新字符串和替换执行次数的两元组。

```
#!python
>>> p = re.compile( '(blue|white|red)')
>>> p.subn( 'colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn( 'colour', 'no colours at all')
('no colours at all', 0)
```

空匹配只有在它们没有紧挨着前一个匹配时才会被替换掉。

```
#!python
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b-d-'
```

如果替换的是一个字符串，任何在其中的反斜杠都会被处理。"\n" 将会被转换成一个换行符，"\r"转换成回车等等。未知的转义如 "\j" 则保持原样。逆向引用，如 "\6"，被 RE 中相应的组匹配而被子串替换。这使你可以在替换后的字符串中插入原始文本的一部分。

这个例子匹配被 "{" 和 "}" 括起来的单词 "section"，并将 "section" 替换成 "subsection"。

```
#!python
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First} section{second}')
'subsection{First} subsection{second}'
```

还可以指定用 (?P<name>...) 语法定义的命名组。"\g<name>" 将通过组名 "name" 用子串来匹配，并且 "\g<number>" 使用相应的组号。所以 "\g<2>" 等于 "\2"，但能在替换字符串里含义不清，如 "\g<2>0"。（"\20" 被解释成对组 20 的引用，而不是对后面跟着一个字母 "0" 的组 2 的引用。）

```
#!python
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

替换也可以是一个甚至给你更多控制的函数。如果替换是个函数，该函数将会被模式中每一个不重复的匹配所调用。在每次调用时，函数会被传入一个 `MatchObject` 的对象作为参数，因此可以用这个对象去计算出替换字符串并返回它。

在下面的例子里，替换函数将十进制翻译成十六进制：

```
#!python
>>> def hexrepl( match ):
...     "Return the hex string for a decimal number"
...     value = int( match.group() )
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

当使用模块级的 re.sub() 函数时，模式作为第一个参数。模式也许是一个字符串或一个 `RegexObject` ；如果你需要指定正则表达式标志，你必须要么使用 `RegexObject` 做第一个参数，或用使用模式内嵌修正器，如 sub("(?i)b+", "x", "bbbb BBBB") returns 'x x'。

常见问题

正则表达式对一些应用程序来说是一个强大的工具，但在有些时候它并不直观而且有时它们不按你期望的运行。本节将指出一些最容易犯的常见错误。

使用字符串方式

有时使用 re 模块是个错误。如果你匹配一个固定的字符串或单个的字符类，并且你没有使用 re 的任何象 IGNORECASE 标志的功能，那么就没有必要使用正则表达式了。字符串有一些方法是对固定字符串进行操作的，它们通常快很多，因为它们都是一个个经过优化的 C 小循环，用以代替大的、更具通用性的正则表达式引擎。

举个 用一个固定字符串替换另一个 的例子，如：你可以把 "deed" 替换成 "word"。re.sub() 似乎正是胜任这个工作的函数，但还是考虑考虑 replace() 方法吧。注意 replace() 也可以在单词里面进行替换，可以把 "swordfish" 变成 "sdeedfish"。不过 RE 也是可以做到的。（为了避免替换单词的一部分，模式将写成 \bword\b，这是为了要求 "word" 两边有一个单词边界。这是个超出 replace 能力的工作）。

另一个常见任务是从一个字符串中删除单个字符或用另一个字符来替代它。你也许可以用 re.sub('\n',' ', s) 这样来实现，但 translate() 能够实现这两个任务，而且比任何正则表达式操作起来更快。（translate 需要配合 string.maketrans 使用。例如：import string 后 'a1b3'.translate(string.maketrans('ab', 'cd'))）

总之，在使用 re 模块之前，先考虑一下你的问题是否可以用更快、更简单的字符串方法来解决。

match() vs search()

match() 函数只检查 RE 是否在字符串开始处匹配，而 search() 则是扫描整个字符串。记住这一区别是重要的。记住，match() 只报告一次成功的匹配，它将从 0 处开始；如果匹配不是从 0 开始的，match() 将不会报告它。

```
#!python
>>> print re.match('super', 'superstition').span()
(0, 5)
>>> print re.match('super', 'insuperable')
None
```

另一方面，search() 将扫描整个字符串，并报告它找到的第一个匹配。

```
#!python
>>> print re.search('super', 'superstition').span()
(0, 5)
>>> print re.search('super', 'insuperable').span()
(2, 7)
```

有时你可能倾向于使用 re.match()，只在RE的前面部分添加 .* 。请尽量不要这么做，最好采用 re.search() 代替之。正则表达式编译器会对 REs 做一些分析以便可以在查找匹配时提高处理速度。一个那样的分析机会指出匹配的第一个字符是什么；举个例子，模式 Crow 必须从 "C" 开始匹配。分析机可以让引擎快速扫描字符串以找到开始字符，并只在 "C" 被发现后才开始全部匹配。

添加 .* 会使这个优化失败，这就要扫描到字符串尾部，然后回溯以找到 RE 剩余部分的匹配。使用 re.search() 代替。

贪婪 vs 不贪婪

当重复一个正则表达式时，如用 a*，操作结果是尽可能多地匹配模式。当你试着匹配一对对称的定界符，如 HTML 标志中的尖括号时这个事实经常困扰你。匹配单个 HTML 标志的模式不能正常工作，因为 .* 的本质是“贪婪”的

```
#!python
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print re.match('<.*>', s).span()
(0, 32)
>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>
```

RE 匹配 在 " `<html>` " 中的 "<"，.* 消耗掉字符串的剩余部分。在 RE 中保持更多的左，虽然 > 不能匹配在字符串结尾，因此正则表达式必须一个字符一个字符地回溯，直到它找到 > 的匹配。最终的匹配从 "<html" 中的 "<" 到 "</title>" 中的 ">",这并不是你所想要的结果。

在这种情况下，解决方案是使用不贪婪的限定符 *?、+?、?? 或 {m,n}?，尽可能匹配小的文本。在上面的例子里， ">" 在第一个 "<" 之后被立即尝试，当它失败时，引擎一次增加一个字符，并在每步重试 ">"。这个处理将得到正确的结果：

```
#!python
>>> print re.match('<.*?>', s).group()
<html>
```

注意用正则表达式分析 HTML 或 XML 是痛苦的。变化混乱的模式将处理常见情况，但 HTML 和 XML 则是明显会打破正则表达式的特殊情况；当你编写一个正则表达式去处理所有可能的情况时，模式将变得非常复杂。象这样的任务用 HTML 或 XML 解析器。

不用 re.VERBOSE

现在你可能注意到正则表达式的表示是十分紧凑，但它们非常不好读。中度复杂的 REs 可以变成反斜杠、圆括号和元字符的长长集合，以致于使它们很难读懂。

在这些 REs 中，当编译正则表达式时指定 re.VERBOSE 标志是有帮助的，因为它允许你可以编辑正则表达式的格式使之更清楚。

re.VERBOSE 标志有这么几个作用。在正则表达式中不在字符类中的空白符被忽略。这就意味着象 dog | cat 这样的表达式和可读性差的 dog|cat 相同，但 [a b] 将匹配字符 "a"、"b" 或空格。另外，你也可以把注释放到 RE 中；注释是从 "#" 到下一行。当使用三引号字符串时，可以使 REs 格式更加干净：

```
#!python
pat = re.compile(r"""
\s*                 # Skip leading whitespace
(?P<header>[^:]+)   # Header name
\s* :               # Whitespace, and a colon
(?P<value>.*?)      # The header's value -- *? used to
# lose the following trailing whitespace
\s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

这个要难读得多：

```
#!python
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

反馈

正则表达式是一个复杂的主题。本文能否有助于你理解呢？哪些部分是否不清晰，或在这儿没有找到你所遇到的问题？如果是那样的话，请将建议发给作者以便改进。

描述正则表达式最全面的书非Jeffrey Friedl 写的《精通正则表达式》莫属，该书由O'Reilly 出版。可惜该书只专注于 Perl 和 Java 风格的正则表达式，不含任何 Python 材料，所以不足以用作Python编程时的参考。（第一版包含有 Python 现已过时的 regex 模块，自然用处不大）。

《精通正则表达式》第三版已经有部分正则表达式使用python说明，另外PHP风格的更是独立一个章节说明。--why？